

## **NCC Group Whitepaper**

# **A Tour of Curve25519 in Erlang**

**Author**

Eric Schorn

### **Abstract**

This whitepaper provides an introduction to elliptic curve cryptography for casual users while delivering a brief tour of the Erlang language for those interested in functional programming. Elliptic curve theory is presented via a simple, ground-up introduction to Curve25519 which culminates in the Diffie–Hellman process for establishing a shared secret in the presence of an eavesdropper. Functional programming is presented by putting this theory into practice via 100 lines of Erlang implementation followed by a script-based demonstration. The reader will simultaneously develop a clearer view of elliptic curve cryptography theory that is also connected to a functional Erlang implementation in practice.



## Introduction

This whitepaper provides an introduction to elliptic curve cryptography for casual users while delivering a brief tour of the Erlang language for those interested in functional programming. Elliptic curve theory is presented via a simple, ground-up introduction to Curve25519 which culminates in the Diffie–Hellman process for establishing a shared secret in the presence of an eavesdropper. Functional programming is presented by putting this theory into practice via 100 lines of Erlang implementation followed by a script-based demonstration. The reader will simultaneously develop a clearer view of elliptic curve cryptography theory that is also connected to a functional Erlang implementation in practice.

Elliptic curves are increasingly popular in transport layer security (TLS), signatures, certificates, messaging, blockchains<sup>1</sup> and most everything crypto.<sup>2</sup> This is partly due to the potential for shorter keys, better performance, less power and fewer side-channels relative to RSA and other alternatives. Unfortunately, even the well-informed person can find the theory impenetrable, the implementation tricks obscure and the frequent assembly-language routines hard to understand.

At the same time, the Erlang language and runtime system make some very different design choices relative to better known alternatives such as JavaScript, Golang, Python, Rust and the C\* family. Erlang is a functional programming language that targets highly-concurrent, distributed, fault-tolerant and non-stop applications – in fact, the code can be upgraded while the application continues to run. While it is instructive to consider such a different paradigm, the typical “hello world” example followed by a review of language syntax and operators does not match everyone’s learning style or available time-commitment.

This whitepaper has a strong focus on simplicity, clarity and brevity. The full source code file is available in [Appendix: Complete Erlang Code Listing on page 15](#) and the terminology is generally consistent with RFC 7748: Elliptic Curves for Security<sup>3</sup> and RFC 8446: TLS 1.3.<sup>4</sup> References are included to both provide additional context as well as support deeper investigation into optimizations, corner cases and side-channel resistance. Note that the presented source code is strictly educational, is not constant-time and it should not be used as-is in production.

## Erlang

Let’s jump right into some Erlang. The biggest challenge to highly-concurrent systems<sup>5</sup> is the sharing of resources, particularly data-structures in memory. Erlang essentially shares nothing and considers an application to be a collection of self-contained message-passing functions without side-effects or the concept of global variables. Further, enigmatic race conditions can occur when reads and writes competing for the same resource happen in differing orders. Largely for this reason, Erlang variables are strictly write-once,<sup>6</sup> which is similar but even more extreme than Rust’s concept of ownership and lifetimes. As this approach precludes loops, recursion is heavily utilized instead, ideally using tail-recursion to minimize stack frame allocation. Process<sup>7</sup> creation is very inexpensive and the runtime system comes with its own scheduler able to juggle ten thousand concurrent processes running on multiple cores as the typical condition. Separate nodes can be further joined together into larger distributed systems<sup>8</sup> consisting of workers, supervisors and application trees. Readers are encouraged to investigate the official Erlang documentation<sup>9</sup> for more insight into OTP components, generic behaviors and hot code-updating.

<sup>1</sup><https://en.bitcoin.it/wiki/Secp256k1>

<sup>2</sup><https://ianix.com/pub/curve25519-deployment.html>

<sup>3</sup><https://tools.ietf.org/pdf/rfc7748.pdf>

<sup>4</sup><https://tools.ietf.org/html/rfc8446#page-96>

<sup>5</sup>[https://en.wikipedia.org/wiki/Concurrent\\_computing](https://en.wikipedia.org/wiki/Concurrent_computing)

<sup>6</sup>[http://erlang.org/doc/reference\\_manual/expressions.html#variables](http://erlang.org/doc/reference_manual/expressions.html#variables)

<sup>7</sup>[http://erlang.org/doc/reference\\_manual/processes.html](http://erlang.org/doc/reference_manual/processes.html)

<sup>8</sup>[http://erlang.org/doc/reference\\_manual/distributed.html](http://erlang.org/doc/reference_manual/distributed.html)

<sup>9</sup><https://erlang.org/doc/search/>

## File Prologue

Each Erlang file must have a few required metadata terms present, specifically the module name and function export list. The module<sup>10</sup> name must match the file name and the functions visible to other modules must be explicitly exported as shown on line 4 below. Erlang is very particular in that a precise function name includes its number of parameters<sup>11</sup> (e.g. `arity` is the trailing `/2` and `/3` below for example). A list consists of comma-separated elements enclosed in `[]`. Finally, the language allows for additional and arbitrary metadata (such as `-author` and `-spec`) that can be picked up by other tools besides the compiler.

In the code below, placing defined names in all caps is strictly by convention, the defined constant values are written as `base#value`<sup>12</sup> integers where `base` is in the range 2 through 36, and functions/statements must end with a period. Each of the specified constants will be described later in their particular usage context.

```
1 -module(curve25519).
2 -author("Eric Schorn").
3
4 -export([mul_k_u/2, test_k_u_iter/3]).
5
6 -define(PRIME, 16#7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED).
7 -define(K_AND, 16#7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8).
8 -define(K_OR, 16#4000000000000000000000000000000000000000000000000000000000000000).
9 -define(U_AND, 16#7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF).
10 -define(ALL256, 16#FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF).
11 -define(A24, 121665).
```

## Modular Arithmetic

Cryptography is full of modular arithmetic<sup>13</sup> and this paper is no different. Think of operands and results as being defined over a set of integers that wrap around. This is similar to a clock's time that wraps around after 24 hours, a video game's score that wraps around after 100,000 points or even an American roulette wheel that wraps around on its 38 individually-numbered pockets. The situation here is most like a novel roulette wheel where multiplicative operations are used to wildly bounce a ball around – the ball being the integer coordinates of a point on a curve.

A clock, game or roulette wheel that wraps around on an even integer presents an issue. If our bouncing ball starts on an *even* integer and that integer is multiplied by any other integer, the result can only be another *even* integer, making the odd integers unreachable. In fact, the same issue applies to schemes that wrap around any number with integer factors; the factors can cause certain destinations to be unreachable by our multiplicative operations. For this reason, a prime modulus<sup>14</sup> (e.g. a wrap-around number without any factors) is often used to make sure all the possibilities remain in play. A ring<sup>15</sup> is loosely defined as this set of possible numbers along with two binary operators that generate them. Some curves are also defined over a prime power  $p^n$  which  $p$  is a prime and  $n > 1$ . For Curve25519, such a huge prime modulus is used that its size roughly approaches the number of atoms in the observable universe. This is the first constant `PRIME` defined on line 6 above and used as `?PRIME` on lines 16, 21 and 26 in the subsequent code below.

Below are three basic functions in Erlang that support modular arithmetic. They each perform the intuitive arithmetic operation followed by a clause that drives the wrap-around behavior. The function name and parameters are first

<sup>10</sup>[http://erlang.org/doc/reference\\_manual/modules.html](http://erlang.org/doc/reference_manual/modules.html)

<sup>11</sup>[http://erlang.org/doc/reference\\_manual/functions.html](http://erlang.org/doc/reference_manual/functions.html)

<sup>12</sup>[http://erlang.org/doc/reference\\_manual/data\\_types.html](http://erlang.org/doc/reference_manual/data_types.html)

<sup>13</sup>[https://en.wikipedia.org/wiki/Modular\\_arithmetic](https://en.wikipedia.org/wiki/Modular_arithmetic)

<sup>14</sup><https://crypto.stanford.edu/pbc/notes/numbertheory/gen.html>

<sup>15</sup><http://mathworld.wolfram.com/Ring.html>

declared and the `->` separator then precedes the implementation, which ends with a period. In each case shown, the implementation consists of a single expression whose value is returned. Comments are required to begin with `%`, though a more common style starts with `%`. The `rem` operator<sup>16</sup> used in each function denotes the remainder after integer division. Variable names must start in upper case.

```

14 -spec(add(X :: non_neg_integer(), Y :: non_neg_integer()) -> non_neg_integer()).
15 add(X, Y) ->
16   (X + Y) rem ?PRIME.
17
18
19 -spec(sub(X :: non_neg_integer(), Y :: non_neg_integer()) -> non_neg_integer()).
20 sub(X, Y) ->   %% Y - Y = 0 = PRIME   so   -Y = PRIME - Y
21   (X + ?PRIME - Y) rem ?PRIME.
22
23
24 -spec(mul(X :: non_neg_integer(), Y :: non_neg_integer()) -> non_neg_integer()).
25 mul(X, Y) ->
26   (X * Y) rem ?PRIME.

```

Erlang is a dynamically typed language, so the `-spec`<sup>17</sup> metadata directives seen preceding each function declaration are for an external type checker called Dialyzer.<sup>18</sup> Consider these lines as comments that formally declare a function's input and output types. Strict type checking is a separate development step and irrelevant at runtime (unless things fail of course).

Again, the three functions above look like ordinary arithmetic with the exception of the `rem ?PRIME` clause. This clause divides an interim result by the prime modulus and returns the remainder to enforce the wrap-around behavior. In modular arithmetic, the modulus `?PRIME` is considered equivalent (or congruent) to zero,<sup>19</sup> and this fact helps recast the subtraction logic to always return an unsigned positive result. Since Erlang has arbitrary precision integers, there is no need to worry about overflow. Also, as indicated by `-spec`, all function parameters in this code are intended to be non-negative integers as are all intermediate values.

It is clear at first glance that the addition and subtraction functions are related in the sense that they reverse each other. Any number can be added to an initial value, that number subsequently subtracted from the sum, and the initial value reappears. This works even in the presence of the `?PRIME` modulus wrapping behavior. For all values `X` there is an additive inverse value `Y`, such that  $X - Y = 0$ . As evident in the `sub()` code above, `-Y` is calculated as simply `PRIME - Y` and is thus always a non-negative integer. However, things are not as simple for the multiplication function's reverse sibling function. Here, a function is needed that returns the multiplicative inverse such that a value times its inverse equals one. A multiplication table mod 19 is shown below, where the entries containing **1** are significant to this goal. The prime modulus 19 was chosen as an example that nicely fits the printed page. Who would have guessed that the square root of 6 mod 19 is both 5 and 14 (since  $5*5$  and  $14*14 \bmod 19$  both equal 6)?!

<sup>16</sup>[http://erlang.org/doc/reference\\_manual/expressions.html#arithmetic-expressions](http://erlang.org/doc/reference_manual/expressions.html#arithmetic-expressions)

<sup>17</sup>[http://erlang.org/doc/reference\\_manual/typespec.html#specifications-for-functions](http://erlang.org/doc/reference_manual/typespec.html#specifications-for-functions)

<sup>18</sup>[http://erlang.org/doc/apps/dialyzer/dialyzer\\_chapter.html](http://erlang.org/doc/apps/dialyzer/dialyzer_chapter.html)

<sup>19</sup><https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/congruence-modulo>

| X  | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| 1  | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 2  | 0 | 2  | 4  | 6  | 8  | 10 | 12 | 14 | 16 | 18 | 1  | 3  | 5  | 7  | 9  | 11 | 13 | 15 | 17 |
| 3  | 0 | 3  | 6  | 9  | 12 | 15 | 18 | 2  | 5  | 8  | 11 | 14 | 17 | 1  | 4  | 7  | 10 | 13 | 16 |
| 4  | 0 | 4  | 8  | 12 | 16 | 1  | 5  | 9  | 13 | 17 | 2  | 6  | 10 | 14 | 18 | 3  | 7  | 11 | 15 |
| 5  | 0 | 5  | 10 | 15 | 1  | 6  | 11 | 16 | 2  | 7  | 12 | 17 | 3  | 8  | 13 | 18 | 4  | 9  | 14 |
| 6  | 0 | 6  | 12 | 18 | 5  | 11 | 17 | 4  | 10 | 16 | 3  | 9  | 15 | 2  | 8  | 14 | 1  | 7  | 13 |
| 7  | 0 | 7  | 14 | 2  | 9  | 16 | 4  | 11 | 18 | 6  | 13 | 1  | 8  | 15 | 3  | 10 | 17 | 5  | 12 |
| 8  | 0 | 8  | 16 | 5  | 13 | 2  | 10 | 18 | 7  | 15 | 4  | 12 | 1  | 9  | 17 | 6  | 14 | 3  | 11 |
| 9  | 0 | 9  | 18 | 8  | 17 | 7  | 16 | 6  | 15 | 5  | 14 | 4  | 13 | 3  | 12 | 2  | 11 | 1  | 10 |
| 10 | 0 | 10 | 1  | 11 | 2  | 12 | 3  | 13 | 4  | 14 | 5  | 15 | 6  | 16 | 7  | 17 | 8  | 18 | 9  |
| 11 | 0 | 11 | 3  | 14 | 6  | 17 | 9  | 1  | 12 | 4  | 15 | 7  | 18 | 10 | 2  | 13 | 5  | 16 | 8  |
| 12 | 0 | 12 | 5  | 17 | 10 | 3  | 15 | 8  | 1  | 13 | 6  | 18 | 11 | 4  | 16 | 9  | 2  | 14 | 7  |
| 13 | 0 | 13 | 7  | 1  | 14 | 8  | 2  | 15 | 9  | 3  | 16 | 10 | 4  | 17 | 11 | 5  | 18 | 12 | 6  |
| 14 | 0 | 14 | 9  | 4  | 18 | 13 | 8  | 3  | 17 | 12 | 7  | 2  | 16 | 11 | 6  | 1  | 15 | 10 | 5  |
| 15 | 0 | 15 | 11 | 7  | 3  | 18 | 14 | 10 | 6  | 2  | 17 | 13 | 9  | 5  | 1  | 16 | 12 | 8  | 4  |
| 16 | 0 | 16 | 13 | 10 | 7  | 4  | 1  | 17 | 14 | 11 | 8  | 5  | 2  | 18 | 15 | 12 | 9  | 6  | 3  |
| 17 | 0 | 17 | 15 | 13 | 11 | 9  | 7  | 5  | 3  | 1  | 18 | 16 | 14 | 12 | 10 | 8  | 6  | 4  | 2  |
| 18 | 0 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  |

Figure 1: Multiplication Mod 19

First, a note on reading the table. Consider any particular entry: the leftmost row header in blue represents a multiplier operand while the topmost column header in blue represents the multiplicand operand, and the intersecting entry contains the multiplied result mod 19. The most interesting entries are **1** because that identifies two operands (leftmost and topmost) that are the inverse of each other – when multiplied their result is 1. There is no simple and straightforward pattern to the location of **1**'s and thus no simple and computationally cheap way of finding a value's inverse. A little complexity enters here based on Fermat's little theorem<sup>20</sup> which states that (when  $p$  is a prime):

$$x^p \equiv x \pmod{p}$$

So if each side is multiplied by  $x^{-2}$  the very useful result is:

$$x^{p-2} \equiv x^{-1} \pmod{p}$$

Note that the congruence<sup>21</sup> relation is used above (rather than strict equality), which means each side has the identical remainder after being divided by the modulus. This is the key to our operating definition of a reverse sibling to the multiplication function. If an operand X is exponentiated to the power of PRIME-2 as on the left side of the equation, the result is the multiplicative inverse of X as on the right side of the equation.

Exponentiation is just repeated multiplication (in the same way that multiplication is just repeated addition, a useful fact used later on). However, the value of the exponent (PRIME-2) presents an issue because its large size precludes a straightforward loop of one-after-another multiplications. Iterating through one multiplication at a time would take (literally) forever. However, consider the following facts:

$$x^2 = x \cdot x \text{ and } x^4 = (x^2)^2 \text{ and } x^8 = ((x^2)^2)^2 \text{ and } x^{16} = (((x^2)^2)^2)^2 \text{ and } x^{31} = x \cdot x^2 \cdot x^4 \cdot x^8 \cdot x^{16}$$

<sup>20</sup><http://mathworld.wolfram.com/FermatsLittleTheorem.html>

<sup>21</sup>[https://en.wikipedia.org/wiki/Modular\\_arithmetic#Congruence\\_classes](https://en.wikipedia.org/wiki/Modular_arithmetic#Congruence_classes)

So, a continued process of squaring operand  $X$  can deliver  $x^{2^{256}}$  in only 255 operations as well as generate all the interim powers of two in the process. Consider a routine that repeatedly squares the operand  $X$  while inspecting each successive bit of the exponent value from the rightmost LSB to the leftmost MSB. When the individual exponent bit of interest is one, the routine multiplies the current square into an interim result. The end result after iterating through the entire exponent is then the inverse of operand  $X$ . Here is the Erlang code for the inverse function. Note that it is using our custom `mul()` function that takes the remainder after each step to prevent the operand's magnitude from exploding.

```

29 -spec(inv(X :: non_neg_integer(), Exp :: non_neg_integer(), Result :: non_neg_integer())
30     -> non_neg_integer()).
31 inv(_X, 0, Result) ->
32     Result;
33
34 inv(X, Exp, Result) ->
35     if
36         Exp band 1 == 1->
37         R1 = mul(Result, X);
38         true ->
39         R1 = Result
40     end,
41     X1 = mul(X, X),
42     inv(X1, Exp bsr 1, R1).

```

In Erlang, it is not unusual to see multiple versions of the same function. When called, the system will attempt to match the function name, arity and operands with each declaration in order and then execute the first match. If the above collection of `inv()` functions were invoked with the middle operand set to `0`, the topmost declaration on line 31 would match and execute. Otherwise, the second declaration on line 34 matches and executes. Note how expressions are separated by a comma, function clauses are separated by a semicolon and both function declaration/bodies together form a single unit ending with a period. The if statement starting on line 35 is similarly a series of clauses that are attempted to match in order, which is why the final `true` value represents the traditional `else` clause, as it matches everything left over. The `_X` parameter in the first `inv()` function declaration on line 31 tells the compiler that this placeholder parameter will not be used.

The lack of looping in Erlang forces a recursive approach. The second portion of the `inv()` function above multiplies the operand into a working result if the exponent LSB=1 (lines 36-37), repeatedly squaring the operand (line 41), and then recursively calls itself with an exponent right shifted by 1 on line 42. The binary shift right operator is denoted `bsr` which helps step through the exponent bits, while `band` is the binary-and operator which helps inspect the LSB value. When the repeatedly-shifted exponent reaches zero, the top function declaration is matched on line 31 and the result finally returned. Line 42 in the second function is an example of tail-call recursion where no further calculation is done so as to obviate the need for a fresh stack frame to be allocated on each iteration.

The above function(s) could easily be tested by taking random values, multiplying each by the calculated inverse value and checking for a result of 1. The multiply and inverse functions are a great example of how some reversible functions can be far easier to run in one direction relative to the reverse direction. That difference will get even more extreme shortly.

## Curve25519

General elliptic curves are specified in a number of forms including:

- The short Weierstrass equation  $y^2 = x^3 + ax + b$  where  $4a^3 + 27b^2$  is nonzero.

- The Edwards equation  $x^2 + y^2 = 1 + dx^2y^2$  where  $d(1 - d)$  is nonzero.
- The Montgomery equation  $By^2 = x^3 + Ax^2 + x$  where  $B(A^2 - 4)$  is nonzero.

The forms above are just a jumping-off point to a tremendous amount of variety and complexity. The nonzero requirements in each form prevents subsequent calculations from reaching non-singular scenarios. NIST has specified a large collection of curves and parameters in the draft NIST SP 800-186 Recommendations for Discrete Logarithm-Based Cryptography: Elliptic Curve Domain Parameters,<sup>22</sup> including Curve25519. As an aside, Bitcoin uses a curve called `secp256k1`.<sup>23</sup> This diversity is also a source of significant controversy within the cryptographic community related to choices of curve forms and constants.<sup>24</sup> One result of these ‘arguments’ is the rapidly increasing adoption of Curve25519.

Curve25519 is a specific Montgomery equation with certain A and B constants, and defined over a particular field of integers. When a point on the curve is referenced, its coordinates represent a solution to the curve equation. For consistency, the coordinates will be labelled  $u$  and  $v$  as used in RFC 7748. The precise Curve25519 equation written in its Montgomery form is:

$$v^2 = u^3 + 486662 \cdot u^2 + u \quad \text{mod } 2^{255} - 19$$

Because curve points can be added, multiplied by a scalar and a few other properties hold, they form a group.<sup>25</sup> The number of discrete points on our ‘novel roulette wheel’ (e.g. elliptic curve), which is known as the curve order, is  $8 \cdot l$  where  $l$  is  $2^{252} + 2774231777372353535851937790883648493$ . Setting aside integers, modular arithmetic and the prime modulus for a moment, the figure below depicts a portion of the curve plotted over real numbers:

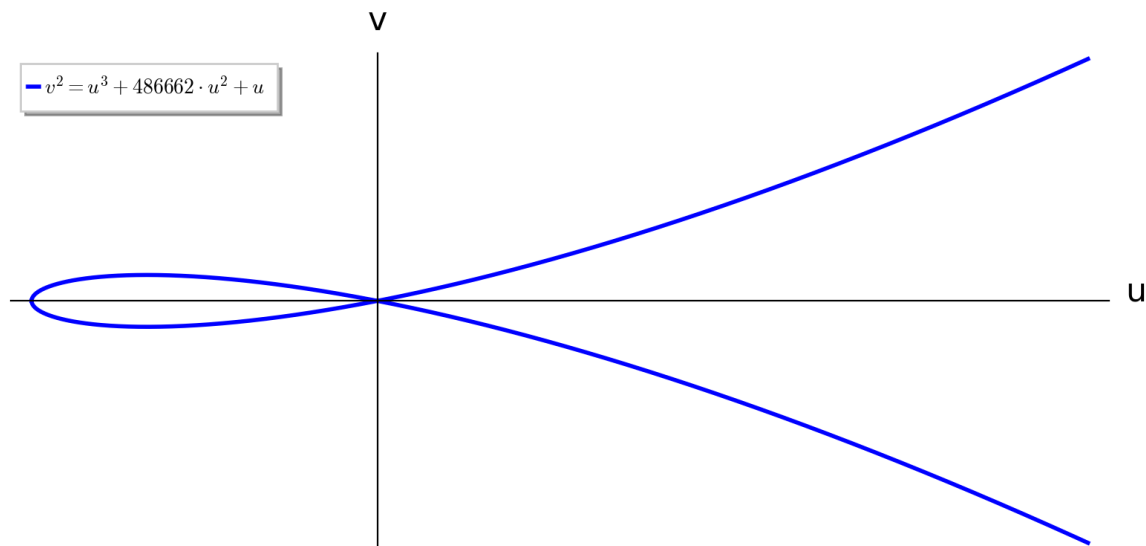


Figure 2: Elliptic Curve Over Real Numbers

Now let’s define an operation called point addition that really just moves from one pair of points (or curve solutions) to another point (or curve solution). Even for the case of real numbers, if the initial points are rational numbers, then the

<sup>22</sup><https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186-draft.pdf>

<sup>23</sup><https://en.bitcoin.it/wiki/Secp256k1>

<sup>24</sup><https://bada55.cr.yt.to/bada55-20150927.pdf>

<sup>25</sup>[https://en.wikipedia.org/wiki/Group\\_theory](https://en.wikipedia.org/wiki/Group_theory)

result will be rational as well. The same is true for our ring of integers. Thus, these calculations will apply to our actual code despite our working with real numbers for the moment. For simplicity, the 'point at infinity' will be ignored.<sup>26</sup>

Here is how point addition works. Start with two points  $P = (u_1, v_1)$  and  $Q = (u_2, v_2)$ . Draw a line between them and derive the equation for that line. That line will intersect the curve in a third point called  $R'$  at  $(u_3, -v_3)$ . Now, the original curve equation, the derived line equation and the two original point solutions are sufficient to solve for the coordinates of  $R'$  as the third solution. The final point  $R$  sum results from just negating the calculated  $v_3$  coordinate of  $R'$ , which effectively flips it across the X axis. A graphical representation of this procedure is shown below.

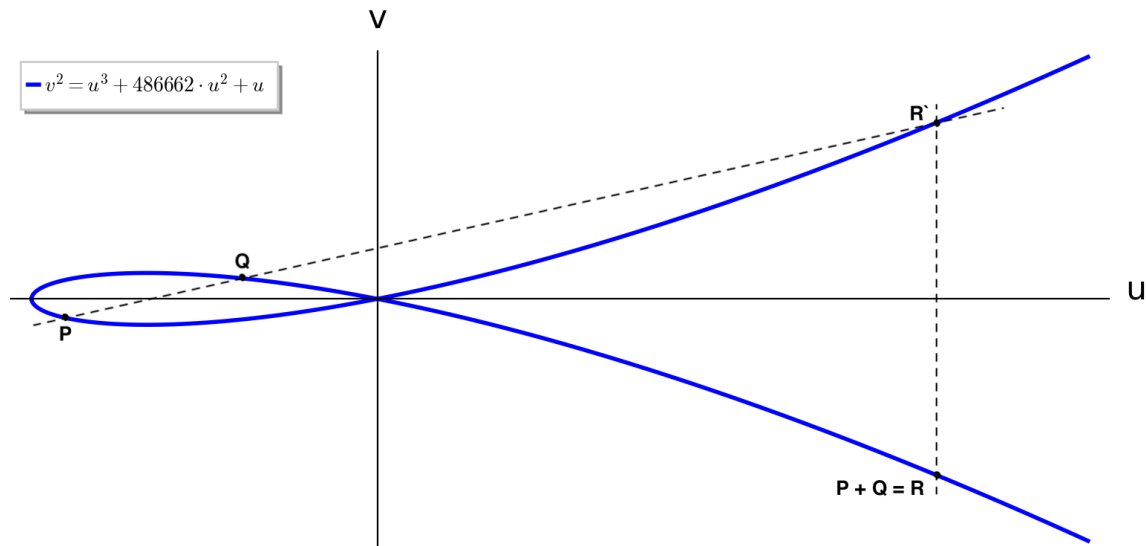


Figure 3: Point Addition on an Elliptic Curve

The solution to the coordinates of R in equation form is:

$$u_3 = \frac{(u_2v_1 - u_1v_2)^2}{u_1u_2(u_2 - u_1)^2}$$

$$y_3 = \frac{(2u_1 + u_2 + 486662)(v_2 - v_1)}{u_2 - u_1} - \frac{(v_2 - v_1)^3}{(u_2 - u_1)^3} - v_1$$

If points P and Q were the same point, then the intersecting line equation would be tangent to the curve at point P and this is now called point doubling with slightly more elaborate math driving different equation results shown below. The general idea works exactly the same as above.

For the point doubling case, we get  $R(u_3, v_3)$  as:

$$u_3 = \frac{(u_1^2 - 1)^2}{4u_1(u_1^2 + 486662u_1 + 1)}$$

$$v_3 = \frac{(2u_1 + u_1 + 486662)(3u_1^2 + 2 \cdot 486662u_1 + 1)}{2v_1} - \frac{(3u_1^2 + 2 \cdot 486662u_1 + 1)^3}{(2v_1)^3} - v_1$$

<sup>26</sup><https://math.stackexchange.com/questions/1118838/elliptic-curve-point-at-infinity>



While the above result looks more complicated rather than less complicated, note how  $u$  can be fully calculated without  $v$ . The doubling calculation can be repeated without involving  $v$ , and the  $v$  value later resolved by the original curve equation if/when needed, meaning we can operate on a point by using only its  $u$  coordinate.

Recall how integer exponentiation was just repeated integer multiplication. Now, point multiplication is just repeated point addition. To be very clear, in this case we are multiplying a scalar times a point, not one point times another point. The same coordinate squaring strategy for exponentiation translates into point doubling strategy for point multiplication. So the same style of calculation can be run - iterate over each bit of the multiplier operand and add a repeatedly-doubled point into the temporary result as appropriate. The multiplier is a scalar called  $K$  and the multiplicand is a point called  $U$  (with the coordinate  $u$ ). The above procedure is a brief description of the Montgomery ladder<sup>27</sup> algorithm as implemented below.

```

50 -spec(mul_k_u(K :: non_neg_integer(), U :: non_neg_integer()) -> non_neg_integer()).
51 mul_k_u(K, U) -> %% x_1 = u, x_2 = 1, z_2 = 0, x_3 = u, z_3 = 1, swap = 0
52   K1 = binary:decode_unsigned(<<K:256/little-unsigned-integer-unit:1>>),
53   K2 = (K1 band ?K_AND) bor ?K_OR,
54   U1 = binary:decode_unsigned(<<U:256/little-unsigned-integer-unit:1>>),
55   U2 = U1 band ?U_AND,
56   mul_k_u(254, K2, U2, 1, 0, U2, 1, 0).
57
58
59 -spec(mul_k_u(T :: -1..254, _K :: non_neg_integer(), _X_1 :: non_neg_integer(), X_2 ::
60   non_neg_integer(), Z_2 :: non_neg_integer(), X_3 :: non_neg_integer(), Z_3 ::
61   non_neg_integer(), Swap :: 0..1) -> non_neg_integer()).
62 mul_k_u(T, _K, X_1, X_2, Z_2, X_3, Z_3, Swap) when T == -1 ->
63   {X_2a, X_3a} = cswap(Swap, X_2, X_3),           %% (x_2, x_3) = cswap(swap, x_2, x_3)
64   {Z_2a, Z_3a} = cswap(Swap, Z_2, Z_3),         %% (z_2, z_3) = cswap(swap, z_2, z_3)
65   Inverse = inv(Z_2a),                          %% Return x_2 * (z_2^(p - 2))
66   Result = mul(X_2a, Inverse),
67   binary:decode_unsigned(<<Result:256/little-unsigned-integer-unit:1>>);
68
69 mul_k_u(T, K, X_1, X_2, Z_2, X_3, Z_3, Swap) -> %% For t = bits-1 down to 0:
70   K_t = (K bsr T) band 1,                        %% k_t = (k >> t) & 1
71   Swap_a = Swap bxor K_t,                       %% swap ^= k_t
72   {X_2a, X_3a} = cswap(Swap_a, X_2, X_3),       %% (x_2, x_3) = cswap(swap, x_2, x_3)
73   {Z_2a, Z_3a} = cswap(Swap_a, Z_2, Z_3),     %% (z_2, z_3) = cswap(swap, z_2, z_3)
74   Swap_b = K_t,                                 %% swap = k_t
75   A = add(X_2a, Z_2a),                          %% A = x_2 + z_2
76   AA = mul(A, A),                               %% AA = A^2
77   B = sub(X_2a, Z_2a),                          %% B = x_2 - z_2
78   BB = mul(B, B),                               %% BB = B^2
79   E = sub(AA, BB),                              %% E = AA - BB
80   C = add(X_3a, Z_3a),                          %% C = x_3 + z_3
81   D = sub(X_3a, Z_3a),                          %% D = x_3 - z_3
82   DA = mul(D, A),                               %% DA = D * A
83   CB = mul(C, B),                              %% CB = C * B

```

<sup>27</sup><https://eprint.iacr.org/2017/293.pdf>

```

84  XX1 = add(DA, CB),           %%  x_3 = (DA + CB)^2
85  X_3b = mul(XX1, XX1),
86  XX2 = sub(DA, CB),         %%  z_3 = x_1 * (DA - CB)^2
87  XX3 = mul(XX2, XX2),
88  Z_3b = mul(X_1, XX3),
89  X_2b = mul(AA, BB),        %%  x_2 = AA * BB
90  XX4 = mul(?A24, E),        %%  z_2 = E * (AA + a24 * E)
91  XX5 = add(AA, XX4),
92  Z_2b = mul(E, XX5),
93  mul_k_u(T - 1, K, X_1, X_2b, Z_2b, X_3b, Z_3b, Swap_b).

```

As mentioned early on, a specific function name includes its arity. So the first function starting on line 51 above is `mul_k_u/2` and is exported as described at the start. This is a different function than the two that follow with the same name starting on lines 62 and 69, which are both internal private functions. Take note of the various punctuations involved in the Erlang syntax which was described earlier.

Let's start with the very bottom function starting on line 69. It is intended to iterate from  $T=254$  down to  $0$  (inclusive) recursively, where the original  $T$  is specified elsewhere. Note the tighter `-spec` for  $T$  on line 59. This function iterates through each individual bit of the  $K$  operand from left MSB to right LSB, and performs a point doubling and optionally point addition. The exact logic derivation will be skipped as the code is conveniently adapted from the pseudo-code in RFC 7748<sup>28</sup> which remains in the comments on the right for reference. Note that variables are only assigned once which requires a little adaptation but really doesn't present much trouble at all. While the code would clearly benefit from an optimized `squaring` function, it isn't strictly necessary as `mul(x, x)` will suffice, and thus skipped for brevity.

The function at the top starting on line 51 launches the calculation. It decodes  $K$  in the proper endian format then performs AND and OR with two predefined constants to mask and set particular bits as directed by RFC 7748. The point  $U$  also takes a little massaging with one predefined constant to mask the most significant bit of the  $u$  coordinate. Then the bottom function is called on line 56 where  $T$  is set to 254.

When the main recursive (bottom, lines 69-93) function's iterative index  $T$  goes below  $0$ , the middle function on line 62 is matched and executed, which does a little final calculation (including the intriguing inverse operation described earlier) before ultimately returning the final calculated result.

Recall that the above calculations are all performed on discrete integers in practice, rather than the real numbers shown in the smooth and intuitive curves earlier. Shown below is the same curve equation plotted over integers mod  $19$ .<sup>29</sup> As mentioned earlier, starting with a discrete point solution and following the process elaborated above will result in another discrete solution. However, the geometric analogy becomes very difficult to see in the discrete space.

<sup>28</sup><https://tools.ietf.org/html/rfc7748#section-5>

<sup>29</sup><http://www.graui.de/code/ffplot/>

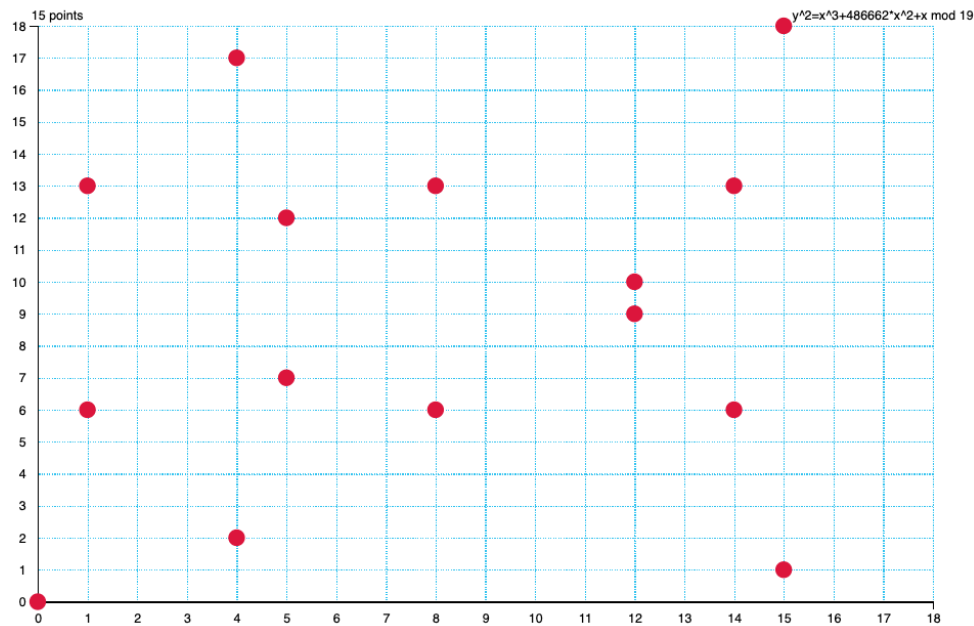


Figure 4: Elliptic Curve Mod 19

In the end, consider the point solution to be like a ball bouncing around a huge roulette wheel in a crazy out-of-order and difficult to reverse fashion. The size of the wheel roughly approaches the number of observable atoms in the universe. The number of wheel spins also roughly approaches to the number of observable atoms in the universe. The resulting point solution pattern is believed to be impossible to work backwards. The (believed to be) intractable task of reversing the point multiplication is known as the elliptic curve discrete logarithm problem<sup>30, 31</sup>.

The fastest known algorithm to solve the elliptic curve discrete logarithm problem runs in time proportional to the square root of the field size. For Curve25519, this size is above  $2^{252}$  thus yielding approximately 128-bits of security. For comparison, NIST suggests a 3072-bit key for comparable security involving RSA algorithms[ABC].

### Cswap

The sharp-eyed reader will spot a single function `cswap()` used earlier that hasn't been defined yet. Here it is for completeness.

```

96 -spec(cswap(Swap :: 0..1, X2 :: non_neg_integer(), X3 :: non_neg_integer()) ->
97   {non_neg_integer(), non_neg_integer()}).
98 cswap(Swap, X2, X3) ->
99   Dummy = Swap*?ALL256 band (X2 bxor X3),
100   X2a = X2 bxor Dummy,
101   X3a = X3 bxor Dummy,
102   {X2a, X3a}.

```

The above function is a conditional swap, and explains the need for the ALL256 constant used on line 99. If the control

<sup>30</sup><https://eprint.iacr.org/2015/1022.pdf>

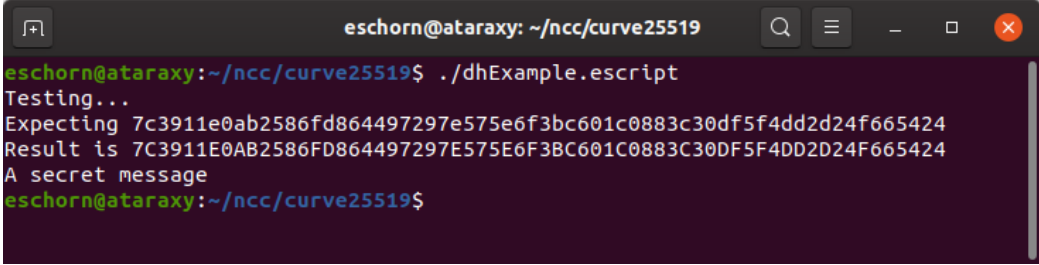
<sup>31</sup><https://andrea.corbellini.name/2015/05/23/elliptic-curve-cryptography-finite-fields-and-discrete-logarithms/>





```
37 io:format("~s~n", [PlainText]). %% <---- Success?
```

The Erlang script runs exactly as expected. After successfully running the 'smoke test', Alice and Bob create their individual private keys, calculate and share the corresponding public keys in the presence of Eve, and finally calculate a shared secret used to (symmetrically) encrypt and decrypt a secret message.



```
eschorn@ataraxy: ~/ncc/curve25519
eschorn@ataraxy:~/ncc/curve25519$ ./dhExample.escript
Testing...
Expecting 7c3911e0ab2586fd864497297e575e6f3bc601c0883c30df5f4dd2d24f665424
Result is 7C3911E0AB2586FD864497297E575E6F3BC601C0883C30DF5F4DD2D24F665424
A secret message
eschorn@ataraxy:~/ncc/curve25519$
```

Figure 5: Secret message encrypted then decrypted under a Diffie-Hellman shared secret

It can be seen that the private keys are kept private and Eve's knowledge of the shared public keys is not sufficient to determine Alice's and Bob's shared secret. Thus, Alice and Bob are able to communicate privately despite the presence of Eve. For clarity, this scenario is not intended to address man-in-the-middle situations which are mitigated by other means; the reader is referred to RFC 8446<sup>33</sup> describing TLS 1.3 for this larger context.

## Conclusion

This whitepaper has introduced elliptical curve cryptography from the ground up in the context of a functional Erlang implementation. All code has been presented and the novel Erlang syntax described. A script demonstrates functionality and the Diffie-Hellman process to establish a shared secret in the presence of an eavesdropper. The reader is now well placed to further investigate both elliptic curve theory and practice, including topics such as constant-time requirements, input validation conditions, alternative algorithm/implementation approaches, optimizations of both point arithmetic and 'big number' field arithmetic, as well as alternative curve equations and their associated arithmetic laws. The reader is further positioned to study additional functional programming concepts in Erlang.

<sup>33</sup><https://tools.ietf.org/html/rfc8446>

The following code is a simple and functional implementation of Curve25519 in Erlang. As this is meant to be a strictly educational example, simplicity is favored over robustness, constant-time characteristics, and checked corner cases.

```
1 -module(curve25519).
2 -author("Eric Schorn").
3
4 -export([mul_k_u/2, test_k_u_iter/3]).
5
6 -define(PRIME, 16#7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED).
7 -define(K_AND, 16#7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8).
8 -define(K_OR, 16#400000000000000000000000000000000000000000000000000000000000000).
9 -define(U_AND, 16#7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF).
10 -define(ALL256, 16#FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF).
11 -define(A24, 121665).
12
13
14 -spec(add(X :: non_neg_integer(), Y :: non_neg_integer()) -> non_neg_integer()).
15 add(X, Y) ->
16     (X + Y) rem ?PRIME.
17
18
19 -spec(sub(X :: non_neg_integer(), Y :: non_neg_integer()) -> non_neg_integer()).
20 sub(X, Y) ->    %% Y - Y = 0 = PRIME    so    -Y = PRIME - Y
21     (X + ?PRIME - Y) rem ?PRIME.
22
23
24 -spec(mul(X :: non_neg_integer(), Y :: non_neg_integer()) -> non_neg_integer()).
25 mul(X, Y) ->
26     (X * Y) rem ?PRIME.
27
28
29 -spec(inv(X :: non_neg_integer()) -> non_neg_integer()).
30 inv(X) ->
31     inv(X, ?PRIME - 2, 1).
32
33
34 -spec(inv(X :: non_neg_integer(), Exp :: non_neg_integer(), Result :: non_neg_integer())
35     -> non_neg_integer()).
36 inv(_X, 0, Result) ->
37     Result;
38
39 inv(X, Exp, Result) ->
40     if
41         Exp band 1 == 1->
42             R1 = mul(Result, X);
43         true ->
44             R1 = Result
45     end,
46     X1 = mul(X, X),
47     inv(X1, Exp bsr 1, R1).
```

```

48
49
50 -spec(mul_k_u(K :: non_neg_integer(), U :: non_neg_integer()) -> non_neg_integer()).
51 mul_k_u(K, U) -> %% x_1 = u, x_2 = 1, z_2 = 0, x_3 = u, z_3 = 1, swap = 0
52   K1 = binary:decode_unsigned(<<K:256/little-unsigned-integer-unit:1>>),
53   K2 = (K1 band ?K_AND) bor ?K_OR,
54   U1 = binary:decode_unsigned(<<U:256/little-unsigned-integer-unit:1>>),
55   U2 = U1 band ?U_AND,
56   mul_k_u(254, K2, U2, 1, 0, U2, 1, 0).
57
58
59 -spec(mul_k_u(T :: -1..254, _K :: non_neg_integer(), _X_1 :: non_neg_integer(), X_2 ::
60   non_neg_integer(), Z_2 :: non_neg_integer(), X_3 :: non_neg_integer(), Z_3 ::
61   non_neg_integer(), Swap :: 0..1) -> non_neg_integer()).
62 mul_k_u(T, _K, _X_1, X_2, Z_2, X_3, Z_3, Swap) when T == -1 ->
63   {X_2a, _X_3a} = cswap(Swap, X_2, X_3),           %% (x_2, x_3) = cswap(swap, x_2, x_3)
64   {Z_2a, _Z_3a} = cswap(Swap, Z_2, Z_3),         %% (z_2, z_3) = cswap(swap, z_2, z_3)
65   Inverse = inv(Z_2a),                            %% Return x_2 * (z_2^(p - 2))
66   Result = mul(X_2a, Inverse),
67   binary:decode_unsigned(<<Result:256/little-unsigned-integer-unit:1>>);
68
69 mul_k_u(T, K, X_1, X_2, Z_2, X_3, Z_3, Swap) -> %% For t = bits-1 down to 0:
70   K_t = (K bsr T) band 1,                          %% k_t = (k >> t) & 1
71   Swap_a = Swap bxor K_t,                          %% swap ^= k_t
72   {X_2a, X_3a} = cswap(Swap_a, X_2, X_3),          %% (x_2, x_3) = cswap(swap, x_2, x_3)
73   {Z_2a, Z_3a} = cswap(Swap_a, Z_2, Z_3),          %% (z_2, z_3) = cswap(swap, z_2, z_3)
74   Swap_b = K_t,                                    %% swap = k_t
75   A = add(X_2a, Z_2a),                              %% A = x_2 + z_2
76   AA = mul(A, A),                                  %% AA = A^2
77   B = sub(X_2a, Z_2a),                              %% B = x_2 - z_2
78   BB = mul(B, B),                                  %% BB = B^2
79   E = sub(AA, BB),                                  %% E = AA - BB
80   C = add(X_3a, Z_3a),                              %% C = x_3 + z_3
81   D = sub(X_3a, Z_3a),                              %% D = x_3 - z_3
82   DA = mul(D, A),                                  %% DA = D * A
83   CB = mul(C, B),                                  %% CB = C * B
84   XX1 = add(DA, CB),                                %% x_3 = (DA + CB)^2
85   X_3b = mul(XX1, XX1),
86   XX2 = sub(DA, CB),                                %% z_3 = x_1 * (DA - CB)^2
87   XX3 = mul(XX2, XX2),
88   Z_3b = mul(X_1, XX3),
89   X_2b = mul(AA, BB),                               %% x_2 = AA * BB
90   XX4 = mul(?A24, E),                               %% z_2 = E * (AA + a24 * E)
91   XX5 = add(AA, XX4),
92   Z_2b = mul(E, XX5),
93   mul_k_u(T - 1, K, X_1, X_2b, Z_2b, X_3b, Z_3b, Swap_b).
94

```



```
95
96 -spec(cswap(Swap :: 0..1, X2 :: non_neg_integer(), X3 :: non_neg_integer()) ->
97   {non_neg_integer(), non_neg_integer()}).
98 cswap(Swap, X2, X3) ->
99   Dummy = Swap*?ALL256 band (X2 bxor X3),
100   X2a = X2 bxor Dummy,
101   X3a = X3 bxor Dummy,
102   {X2a, X3a}.
103
104
105 -spec(test_k_u_iter(K :: non_neg_integer(), U :: non_neg_integer(), Iter ::
106   non_neg_integer()) -> ok).
107 test_k_u_iter(K, U, Iter) when Iter > 0 ->
108   %% For each iteration, set k to be the result of calling the function and
109   %% u to be the old value of k. The final result is the value left in k.
110   K1 = mul_k_u(K, U),
111   U1 = K,
112   test_k_u_iter(K1, U1, Iter - 1);
113
114 test_k_u_iter(K, _U, 0) ->
115   io:fwrite("Result is ~.16B~n", [K]).
116 %% curve25519:test_k_u_iter(
117 %%       16#0900000000000000000000000000000000000000000000000000000000000000,
118 %%       16#0900000000000000000000000000000000000000000000000000000000000000, 1000).
119 %% After 1K iterations: 684cf59ba83309552800ef566f2f4d3c1c3887c49360e3875f2eb94d99532c51
120 %% After 1M iterations: 7c3911e0ab2586fd864497297e575e6f3bc601c0883c30df5f4dd2d24f665424
```